Beer League Beauties – Datamart Overview & How It All Fits Together

This document explains, in plain English, how the **ultimate BLB project** is structured, how the **datamart tables** work, how the **pipeline** runs, and how to plug everything in to **Power BI**.

It combines and extends the explanations from our recent chats.

---

1. High■Level Architecture

At a high level, the project has three main data layers:

1. **Stage (stg_)**
   - Directly ingests raw Excel/CSV game packages plus the BLB master workbook (`BLB_Table
s.xlsx`).
   - Tables look very close to the raw data (events, shifts, xy, shots, video times, dim t
ables).
   - Purpose: *land data with minimal transformation; keep raw-ish but cleaned types and c
olumn names.*

2. **Intermediate (int_)**
   - Performs heavier transformations that are still “technical”:
     - Events long → wide
     - Shifts wide → long → logical player shifts
     - Joining shifts and events by game/time/shift_index
     - Building event chains: `linked_event_index`,`sequence_index`, `play_index`
     - Attaching XY + rink zones
   - Purpose: *create clean, re-usable building blocks for the mart, but not yet optimized
 for analytics.*

3. **Datamart (fact_/dim_)**
   - Star/snowflake schema, one set of tables for *all games*.
   - Each fact table has a clear **grain** (one row = one shift, one player-game, one even
t, etc.)
   - Designed for **Power BI**, **Python dashboards**, and **ML**.

The **Python pipeline** orchestrates movement through these layers:

- `stage` layer: load raw → `stg_*`
- `intermediate` layer: SQL + Python transforms → `int_*`
- `mart` layer: SQL + Python transforms → `fact_*` and `dim_*`, then export CSVs.

All orchestration is controlled through a single main entrypoint (e.g., `main.py` or `run_
stage_only.py`),
with user inputs for which games to process, whether to reload BLB tables, etc.

---

2. Datamart Tables – What Each One Represents

Below is a conceptual data dictionary of the **mart layer**.
Exact column names are defined in the SQL DDL files (e.g., `sql/ddl/02_create_tables.sql`)
, but this describes the semantics.

2.1 Dimension Tables

2.1.1 `dim_player`

- **Grain:** One row per player across all seasons.
- **Key:** `player_id` (from `BLB_Tables`), stable across games.
- **Key columns:**
  - `player_id`
  - `player_full_name`, `first_name`, `last_name`
  - `shoots`, `position_group` (F/D/G), `primary_position`
  - `skill_rating` (2–6)
  - Optional: `team_id_current`, `league_id_current`, etc.

Purpose: Central player dimension. Everything that has a player should be able to join here via `player_id`.

Used for:

- Player cards (single-game and season), microstats, comparisons.
- Rating-aware context (e.g., quality of competition / teammates).

2.1.2 `dim_team`

- **Grain:** One row per team.
- **Key:** `team_id`
- **Key columns:**
  - `team_id`
  - `team_name`, `long_team_name`, `team_cd` (short code)
  - `league_id`, `league`
  - Optional: `team_color1`, `team_color2` for visual encoding

Purpose: Team lookups and grouping.

Used for:

- Team filters, team-level summaries, stacked bar charts by team.
- Coloring schemes in Power BI/Dash (home vs away colors).

2.1.3 `dim_season`

- **Grain:** One row per season.
- **Key:** `season` or `season_id`
- **Key columns:**
  - `season` (e.g., 2024–2025)
  - `season_start_date`, `season_end_date`
  - `league_id`

Purpose: Slice any analysis by season; support multi-season trends.

2.1.4 `dim_schedule`

- **Grain:** One row per game.
- **Key:** `game_id`
- **Key columns:**
  - `game_id`
  - `game_date` (joins to `dim_date.date`)
  - `home_team_id`, `away_team_id`
  - `home_score_final`, `away_score_final`

  - `season`, `league_id`
  - **Tracking metadata:** planned / partially tracked / fully tracked, plus description o
f what data exists (shifts, events, XY, shots, video).

Purpose: “Game header” table. Almost every fact table has `game_id` and therefore can
connect to `dim_schedule`.

Used for:

- Game list pages (ESPN/NHL style views).
- Filtering all visuals to a specific game or range of games.

2.1.5 `dim_rink_zones`

- **Grain:** One row per rink zone or rink segment.
- **Key:** `zone_id`
- **Key columns:**
  - `zone_id`
  - `zone_name` (e.g., High Danger Slot, Left Circle)
  - `x_min`, `x_max`, `y_min`, `y_max` or polygon coordinates
  - `danger_band` (e.g., High, Medium, Low)

Purpose: Map XY coordinates to meaningful zones.

Used for:

- Danger-based shot charts and heatmaps.
- xG models that incorporate shot location/danger band.

2.1.6 `dim_game_players`

- **Grain:** One row per player–game (roster entry).
- **Key:** `game_player_id` (surrogate).
- **Key columns:**
  - `game_player_id`
  - `game_id`
  - `player_id`
  - `team_id`
  - `jersey_number`
  - Optional: `line_role`, `is_starting_goalie`, etc.

Purpose: Bridge between `dim_player` and `dim_schedule` for a specific game.

Used for:

- Game rosters, scratches, subs.
- With/without-you style analysis (who dressed but maybe didn’t record tracked events).

> **Note:** `dim_date` from `BLB_Tables` is also used as a standard calendar dimension, re
lated to `dim_schedule.game_date`.

2.2 Fact Tables

2.2.1 `fact_gameroster`

- **Grain:** One row per player–game with league-provided stats.
- **Key:** Composite (`game_id`, `player_id`, `team_id`) or a surrogate.
- **Columns (examples):**
  - Goals, assists, shots, PIM, plus/minus, basic counting stats.

Purpose: Official stat line for every player in every game, even if manual tracking is partial or missing.

Used for:

- Season-level counting stats.
- Validating manual tracking against league numbers.
- Player classification (e.g., shooter vs playmaker) over the full season.

2.2.2 `fact_shifts` (shift header, team-level)

- **Grain:** One row per *logical shift per team*.
- **Key:** `shift_id` (e.g., `SH<game_id><index>`).
- **Columns (examples):**
  - `game_id`, `team_id`, `period`
  - `shift_index_min`, `shift_index_max` (raw indices from wide data)
  - `shift_start_total_seconds`, `shift_end_total_seconds`
  - `shift_duration_seconds`
  - `shift_effective_seconds` (duration – stoppage time)
  - Score state at start/end and situation/strength
  - **Rating context:** min/avg/max rating of all own skaters and opposing skaters on the ice.

Purpose: High-level shift context for each team.

Used for:

- Shift tempo analysis.
- Joining events by time/period to the shift context.

2.2.3 `fact_shift_players` (player-on-ice shifts)

- **Grain:** One row per player–logical-shift.
- **Key:** `shift_player_id` (e.g., `SP<game_id><player><logical_shift_seq>`).
- **Columns (examples):**
  - `shift_player_id`
  - `shift_id`
  - `game_id`, `team_id`, `player_id`
  - `period`
  - `logical_shift_number` (for that player in the game)
  - `logical_shift_segment_count` (how many raw rows combined)
  - `time_on_ice_seconds`, `time_on_ice_seconds_nostop`
  - Rating context of opposing skaters on ice (min/avg/max).

Purpose: Detailed TOI dataset for players by shift.

Used for:

- All TOI visuals, especially rating-aware matchups.
- Building per-period and game totals, and microstats per shift.

2.2.4 `fact_events` (event header / wide)

- **Grain:** One row per event (event index per game).
- **Key:** `event_id` (e.g., `EV<game_id><event_index>`).
- **Columns (examples):**
  - `game_id`, `event_index`, `period`
  - `event_start_total_seconds`, `event_end_total_seconds`, `event_duration_seconds`
  - `type`, `event_detail_1`, `event_detail_2`
  - `play_index`, `sequence_index`, `linked_event_index`
  - `team_id_for`, `team_id_against`
  - `score_state_start`, `score_state_end`
  - `shift_id_for`, `shift_id_against` (shifts for both teams)
  - Rating context: on-ice min/avg/max rating for each side.

Purpose: Main event table for the game – every play you tracked is represented here.

Used for:

- Timeline visualizations, xG, sequence analysis.
- Driving “what happened when” questions and microstats.

2.2.5 `fact_event_players` (who did what in an event)

- **Grain:** One row per event–role–player.
- **Key:** `event_player_id` (e.g., `EP<game_id><event_index><role>`).
- **Columns (examples):**
  - `event_player_id`
  - `event_id`, `game_id`
  - `player_id`, `team_id`
  - `role_type` (event_player_1/2, opp_player_1/2, etc.)
  - Flags: `is_primary_actor`, `is_assist_primary`, `is_assist_secondary`
  - Faceoff flags: `is_faceoff_winner`, `is_faceoff_loser`
  - Success flags: `event_successful`, `play_successful`

Purpose: Exactly *who* was involved in each event, and in *what role*.

Used for:

- All player-level microstats: shots, passes, entries/exits, giveaways, takeaways, etc.
- Head-to-head matchups, targeted stats (how often a player is opp_player_1).

2.2.6 `fact_box_score` (player–game microstats summary)

- **Grain:** One row per player–game (tracking-enhanced box score).
- **Key:** `box_score_id` (or composite of `game_id`, `player_id`, `team_id`).
- **Columns (examples):**
  - Core stats: goals, primary assists, secondary assists, points
  - Shots, shot attempts, missed shots, blocked shots
  - Giveaways, takeaways, hits (if tracked)
  - Faceoff wins/losses (for faceoff-takers)
  - Advanced: Corsi For/Against, Fenwick For/Against, expected goals (if modeled)
  - Microstats: zone entries/exits, passes attempted/completed by type
  - TOI: per game, per strength, per situation, etc.

Purpose: Game stat line for each player that combines **league data** + **tracking dat
a**.

Used for:

- Player scorecards in Power BI and Dash.
- ML feature sets across many games.

2.2.7 `fact_linked_events`

- **Grain:** One row per linked_event_index chain (e.g., shot → save → rebound → goal).
- **Key:** `linked_chain_id`.
- **Columns (examples):**
  - `game_id`, `linked_event_index`
  - `chain_type` (shot sequence, rebound sequence, etc.)
  - `start_event_id`, `end_event_id`
  - `chain_length`, `duration_seconds`
  - `contains_goal`, `contains_shot_on_goal`
  - Zone / rating context at chain start.

Purpose: Capture tightly linked tactical sequences around a single core event.

Used for:

- Studying rebound plays, save–rebound–goal sequences, etc.
- xG modeling at the “micro play” level.

2.2.8 `fact_sequences`

- **Grain:** One row per `sequence_index` group (e.g., turnover → exit → entry → shot).
- **Key:** `sequence_id`.
- **Columns (examples):**
  - `game_id`, `sequence_index`
  - `sequence_type` (e.g., transition, OZ cycle – can be derived later)
  - `start_event_id`, `end_event_id`
  - `duration_seconds`, number of events
  - Counts of passes, entries, exits, shots
  - Whether the sequence results in a goal.

Purpose: Capture medium-sized plays that link multiple events together.

Used for:

- Understanding which types of sequences drive offense.
- Building features for ML models about play efficiency.

2.2.9 `fact_plays`

- **Grain:** One row per `play_index` group (macro plays).
- **Key:** `play_id`.
- **Columns (examples):**
  - `game_id`, `play_index`
  - `play_type` (rush, cycle, forecheck; can be added later)

  - `start_event_id`, `end_event_id`
  - `duration_seconds`, number of events
  - `play_results_in_goal`, `play_results_in_shot`
  - Rating context for attackers and defenders.

Purpose: High-level “plays” that might span multiple sequences or chains.

Used for:

- Coaching analytics: which patterns of play work best?
- Advanced ML tasks like predicting play outcome probability.

---

3. How the Pipeline Runs (Stage → Intermediate → Mart)

The orchestration is handled by a Python class (e.g., `PipelineOrchestrator`) and executed via `main.py`.

3.1 Logical Steps

For each game (stored under `data/raw/games/<game_id>/`), the pipeline does:

1. **Stage load (stg_)**
   - Read game tracking workbook (`<game_id>_tracking.xlsx`):
     - `events` sheet → `stg_events`
     - `shifts` sheet → `stg_shifts`
   - Read any XY event files: `events/<game_id>_event_*.csv` → `stg_events_xy`
   - Read any shot XY files: `shots/<game_id>_shot_*.csv` → `stg_shots_xy`
   - Read video timing file → `stg_video_times`
   - Read master workbook (`BLB_Tables.xlsx`) → `stg_dim_*` and `stg_fact_gameroster`, etc.
   - Every stage table is **normalized** (lowercase snake_case) and gets an `update_date` datestamp.

2. **Intermediate transforms (int_)**
   - **Shifts wide → long → logical shifts:**
     - Explode wide shift line columns into one row per player per raw shift row.
     - Collapse consecutive `shift_index` rows per player into **logical shifts**, computing:
       - `logical_shift_number`, `segment_count`, start/end times, durations.
     - Compute **TOI with and without stoppage**, using the `stoppage_time` column.
   - **Events long → wide:**
     - Pivot event player roles so each `event_index` becomes one row in wide form.
     - Drop raw columns ending in `_` and keep normalized columns.
   - **Join events ■ shifts:**
     - For each event, find which shifts were on the ice using `game_id`, `period` and total seconds.
   - **Attach ratings:**
     - Join `dim_player` to get `skill_rating` for all players on ice.
     - Compute min/avg/max rating on each side for shifts and events.
   - **Build chains (linked, sequence, play):**
     - Group events by `linked_event_index`, `sequence_index`, and `play_index`.
     - Summarize start/end, counts, durations, and outcomes.
   - **Enrich XY:**
     - Join event XY and shot XY to `fact_events`-level keys via `linked_event_index` & `player_number`/`team` rules.
     - Map XY to `dim_rink_zones` to get danger zones, distances, angles.

3. **Mart builds (fact_*, dim_*)**

- **Dimensions** are populated:
  - `dim_player`, `dim_team`, `dim_schedule`, `dim_game_players`, `dim_rink_zones`, `dim_season`, `dim_date`.
- **Fact tables**:
  - `fact_shifts` and `fact_shift_players` from the logical shift structures.
  - `fact_events` from the event wide header table.
  - `fact_event_players` from event-long data plus roles.
  - `fact_box_score` using aggregations from `fact_event_players`, `fact_shifts`, `fact_gameroster`.
  - `fact_linked_events`, `fact_sequences`, `fact_plays` from chain groupings.
- All fact tables are **global across all games**:
  - New games append new rows using `game_id` in primary keys.
  - If you re-run a specific game, the ETL can delete/re-insert that game's rows for idempotency.

4. **Exports**
- Each datamart table is exported as a CSV into `/output` (or `data/output/`), ready for Power BI.

---

4. Example Deep Dive: Building `fact_events`

Below is a conceptual, step-by-step view of how `fact_events` can be built.
(Exact code is split across SQL + Python modules in the project.)

4.1 Inputs

- `stg_events` (long format from tracking workbook)
- `int_shift_logical` (or equivalent) – logical shifts per team & player
- `stg_events_xy` and `stg_shots_xy` (optional, XY data)
- `dim_player`, `dim_team`, `dim_schedule`

4.2 Step 1 – Clean and normalize events

Pseudo-SQL:

```sql
CREATE TABLE int_events_clean AS
SELECT
    game_id,
    event_index,
    period,
    event_start_total_seconds,
    event_end_total_seconds,
    event_end_total_seconds - event_start_total_seconds AS event_duration_seconds,
    type,
    event_detail_1,
    event_detail_2,
    event_successful,
    play_details_1,
    play_details_2,
    play_successful,
    linked_event_index,
    sequence_index,
    play_index,
    team_for,   -- derived from raw "team" columns
    team_against
FROM stg_events
WHERE event_index IS NOT NULL;
```

```

### 4.3 Step 2 – Attach shift context

We want to know which shifts were on the ice for each event. Conceptually:

```sql
CREATE TABLE int_events_with_shifts AS
SELECT
    e.*,
    s_for.shift_id   AS shift_id_for,
    s_against.shift_id AS shift_id_against
FROM int_events_clean e
LEFT JOIN fact_shifts s_for
  ON  s_for.game_id = e.game_id
  AND s_for.team_id = e.team_for
  AND s_for.period = e.period
  AND e.event_start_total_seconds BETWEEN s_for.shift_start_total_seconds
                                  AND s_for.shift_end_total_seconds
LEFT JOIN fact_shifts s_against
  ON  s_against.game_id = e.game_id
  AND s_against.team_id = e.team_against
  AND s_against.period = e.period
  AND e.event_start_total_seconds BETWEEN s_against.shift_start_total_seconds
                                  AND s_against.shift_end_total_seconds;
```

Now each event knows:

- Which team “for” and “against” was on the ice.
- Which **shift IDs** they correspond to.

4.4 Step 3 – Attach on-ice rating context

From `fact_shift_players` we can aggregate on-ice ratings:

```sql
CREATE TABLE int_events_with_ratings AS
WITH on_ice_for AS (
    SELECT
        sp.game_id,
        sp.shift_id,
        MIN(p.skill_rating) AS rating_for_min,
        AVG(p.skill_rating) AS rating_for_avg,
        MAX(p.skill_rating) AS rating_for_max
    FROM fact_shift_players sp
    JOIN dim_player p ON p.player_id = sp.player_id
    GROUP BY sp.game_id, sp.shift_id
),
on_ice_against AS (
    SELECT
        sp.game_id,
        sp.shift_id,
        MIN(p.skill_rating) AS rating_against_min,
        AVG(p.skill_rating) AS rating_against_avg,
        MAX(p.skill_rating) AS rating_against_max
    FROM fact_shift_players sp
    JOIN dim_player p ON p.player_id = sp.player_id
    GROUP BY sp.game_id, sp.shift_id
)
```
```

```
SELECT
    e.*,
    f.rating_for_min,
    f.rating_for_avg,
    f.rating_for_max,
    a.rating_against_min,
    a.rating_against_avg,
    a.rating_against_max
FROM int_events_with_shifts e
LEFT JOIN on_ice_for f
  ON f.game_id = e.game_id
 AND f.shift_id = e.shift_id_for
LEFT JOIN on_ice_against a
  ON a.game_id = e.game_id
 AND a.shift_id = e.shift_id_against;
```

4.5 Step 4 – Attach XY + Rink Zones (optional)

Use `stg_events_xy` and `dim_rink_zones`:

```sql
CREATE TABLE int_events_with_xy AS
SELECT
    e.*,
    xy.x_coord,
    xy.y_coord,
    rz.zone_id,
    rz.zone_name,
    rz.danger_band
FROM int_events_with_ratings e
LEFT JOIN stg_events_xy xy
  ON xy.game_id = e.game_id
 AND xy.linked_event_index = e.linked_event_index
 AND xy.player_number LIKE 'p%'  -- puck location
LEFT JOIN dim_rink_zones rz
  ON xy.x_coord BETWEEN rz.x_min AND rz.x_max
 AND xy.y_coord BETWEEN rz.y_min AND rz.y_max;
```

4.6 Step 5 – Finalize `fact_events`

Finally, we insert into the final mart table:

```sql
INSERT INTO fact_events (
    event_id,
    game_id,
    event_index,
    period,
    event_start_total_seconds,
    event_end_total_seconds,
    event_duration_seconds,
    type,
    event_detail_1,
    event_detail_2,
    event_successful,
    play_details_1,
    play_details_2,
    play_successful,
```

```
    linked_event_index,
    sequence_index,
    play_index,
    team_id_for,
    team_id_against,
    shift_id_for,
    shift_id_against,
    rating_for_min,
    rating_for_avg,
    rating_for_max,
    rating_against_min,
    rating_against_avg,
    rating_against_max,
    x_coord,
    y_coord,
    zone_id,
    zone_name,
    danger_band
)
SELECT
    CONCAT('EV', e.game_id, LPAD(e.event_index::text, 4, '0')) AS event_id,
    e.game_id,
    e.event_index,
    e.period,
    e.event_start_total_seconds,
    e.event_end_total_seconds,
    e.event_duration_seconds,
    e.type,
    e.event_detail_1,
    e.event_detail_2,
    e.event_successful,
    e.play_details_1,
    e.play_details_2,
    e.play_successful,
    e.linked_event_index,
    e.sequence_index,
    e.play_index,
    e.team_for      AS team_id_for,
    e.team_against  AS team_id_against,
    e.shift_id_for,
    e.shift_id_against,
    e.rating_for_min,
    e.rating_for_avg,
    e.rating_for_max,
    e.rating_against_min,
    e.rating_against_avg,
    e.rating_against_max,
    e.x_coord,
    e.y_coord,
    e.zone_id,
    e.zone_name,
    e.danger_band
FROM int_events_with_xy e;
```

This is not the exact code from your repository, but it mirrors the design: each step is s
mall, testable, and re-runnable.

---

5. Power BI – Relationships & Example DAX

5.1 Recommended Relationships (Star / Snowflake)

Core relationships:

- `dim_schedule[game_id]` →
  - `fact_shifts[game_id]`
  - `fact_shift_players[game_id]`
  - `fact_events[game_id]`
  - `fact_event_players[game_id]`
  - `fact_box_score[game_id]`
  - `fact_gameroster[game_id]`
  - `fact_linked_events[game_id]`
  - `fact_sequences[game_id]`
  - `fact_plays[game_id]`

- `dim_player[player_id]` →
  - `fact_shift_players[player_id]`
  - `fact_event_players[player_id]`
  - `fact_box_score[player_id]`
  - `fact_gameroster[player_id]`

- `dim_team[team_id]` →
  - `dim_schedule[home_team_id]` (inactive relation, used in measures)
  - `dim_schedule[away_team_id]` (inactive relation)
  - `fact_shifts[team_id]`
  - `fact_shift_players[team_id]`
  - `fact_events[team_id_for]` and `fact_events[team_id_against]` (one active, one inactiv
e)
  - `fact_event_players[team_id]`
  - `fact_box_score[team_id]`
  - `fact_gameroster[team_id]`

- `dim_date[date]` → `dim_schedule[game_date]`

Optional:

- `dim_rink_zones[zone_id]` → `fact_events[zone_id]` (one-to-many).

5.2 Example DAX: Game-level summary (ESPN/NHL style)

Assume you are on a **Game Summary** page with `dim_schedule[game_id]` filtered.

Goals – Home Team

```DAX
Goals Home =
CALCULATE (
    SUM ( fact_box_score[goals] ),
    TREATAS ( VALUES ( dim_schedule[home_team_id] ), fact_box_score[team_id] )
)
```

Goals – Away Team (same but `away_team_id`).

Shots on Goal – Home

```DAX
Shots on Goal Home =
```

```DAX
CALCULATE (
    SUM ( fact_box_score[shots_on_goal] ),
    TREATAS ( VALUES ( dim_schedule[home_team_id] ), fact_box_score[team_id] )
)
```

Corsi For – Home (5v5)

```DAX
Corsi For Home 5v5 =
CALCULATE (
    SUM ( fact_box_score[corsi_for] ),
    fact_box_score[strength] = "5v5",
    TREATAS ( VALUES ( dim_schedule[home_team_id] ), fact_box_score[team_id] )
)
```

5.3 Example DAX: Player card statistics

Suppose the Player Card page has `dim_player[player_id]` in the filter context.

Total TOI (minutes)

```DAX
TOI Minutes =
DIVIDE (
    SUM ( fact_box_score[toi_seconds_total] ),
    60.0
)
```

Individual Expected Goals (ixG) – placeholder if you add xG per event:

```DAX
ixG =
SUM ( fact_box_score[xg_for] )
```

Zone Entry Success %

```DAX
Zone Entry Success % =
VAR EntriesAttempted =
    SUM ( fact_box_score[zone_entries_attempted] )
VAR EntriesSuccessful =
    SUM ( fact_box_score[zone_entries_successful] )
RETURN
    DIVIDE ( EntriesSuccessful, EntriesAttempted )
```

5.4 Example DAX: Line combo effectiveness

Using `fact_shift_players` and a derived `Line Id` (e.g., concatenation of three forwards' player_ids):

```DAX
Line Shots For =
SUMX (
    VALUES ( fact_shifts[shift_id] ),
    CALCULATE ( SUM ( fact_events[is_shot_for_flag] ) )
```

)
```

You would create a dimension table `dim_line` to describe each line combo and relate it to an intermediate line fact (e.g., `fact_line_shifts`), but that's a next-stage enhancement
.

---

## 6. How to Run the Pipeline (User Controls)

You control everything through `main.py` (or equivalent) using either:

- **Interactive mode:**

  ```bash
  python main.py
  ```

  You'll be prompted for:

  - Which games to process (all unprocessed vs specific).
  - Whether to reload BLB tables.
  - Whether to rebuild intermediate/mart layers.
  - Whether to export CSVs for Power BI.

- **CLI flags (non-interactive):**

  ```bash
  # Process specific game(s)
  python main.py --games 18969,18970

  # Process all unprocessed games
  python main.py --process-all

  # Reload BLB tables
  python main.py --reload-blb

  # Export mart tables to CSVs (no new processing)
  python main.py --export
  ```

Internally, the orchestrator:

1. Discovers game folders under `data/raw/games/`.
2. Compares to games already in the mart (by `game_id`).
3. Runs stage → intermediate → mart steps for the selected games.
4. Writes logs (with timestamps, modules, and levels) so you can debug issues.
5. Exports datamart tables into a configured `/output` directory as CSVs.

Since all fact tables include `game_id`, the mart holds **all games together**,
and Power BI can slice by any game or set of games.

---

7. How to Use This Document

- Treat this as your **conceptual map** of the BLB project.
- When you're unsure what a table represents, check:
  - Its **grain** (one row = what?)
  ```

- Its **keys** (what is the primary key, how does it relate to dims?)
- Its **purpose** (what analysis is it built to support?)

From here you can:

1. Open the SQL DDLs in `sql/ddl/` and map them to this description.
2. Open the transformation SQL/Python in `sql/transform` and `src/` to see the exact code.
3. Wire up Power BI using the relationships and DAX ideas here to start building:
   - Game summaries (ESPN/NHL style)
   - Player cards (NHL Edge style)
   - Line combo and head-to-head visuals
   - Microstat dashboards inspired by the links you shared.

This document has been saved as TXT, HTML, and PDF for easy reference.